# The Grace Programming Language
# Draft Specification Version 0.5.1853

Andrew P. Black  Kim B. Bruce  James Noble

August 30, 2014

## 1 Introduction

This is a specification of the Grace Programming Language. This specification is notably incomplete, and everything is subject to change. In particular, this version does *not* address:

- 
- the library, especially collections and collection literals
- static type system (although we've made a start)
- module system
- dialects
- metadata (Java's @annotations, C♯ attributes, final, abstract etc)
- immutable data and pure methods.
- reflection
- assertions, data-structure invariants, pre & post conditions, contracts
- regexps?
- libraries, including more (complex?) Numeric types and testing

For discussion and rationale, see http://gracelang.org.

Where this document gives "(options)", we outline choices in the language design that have yet to be made.

## 2 User Model

> *All designers in fact have user and use models consciously or subconsciously in mind as they work. Team design . . . requires explicit models and assumptions.*
>
> Frederick P. Brooks, *The Design of Design*. 2010.

1. First year university students learning programming in CS1 and CS2 classes that are based on object-oriented programming.
   (a) The courses may be structured objects first, or imperative first. Is it necessary to support "procedures first"?
   (b) The courses may be taught using dynamic types, static types, or both in combination (in either order).
   (c) We aim to offer some (but not necessarily complete) support for "functional first" curricula, primarily for courses that proceed rapidly to imperative and object-oriented programming.
2. University students taking second year classes in programming, algorithms and data structures, concurrent programming, software craft, and software design.
3. Faculty and teaching assistants developing libraries, frameworks, examples, problems and solutions, for first and second year programming classes.
4. Programming language researchers needing a contemporary object-oriented programming language as a research vehicle.
5. Designers of other programming or scripting languages in search of a good example of contemporary OO language design.

# 3 Syntax

> *Much of the following text assumes the reader has a minimal grasp of computer terminology and a "feeling" for the structure of a program.*
>
> Kathleen Jensen and Niklaus Wirth, *Pascal: User Manual and Report.*

Grace programs are written in Unicode. Reserved words are written in the ASCII subset of Unicode. As a matter of policy, the names of methods defined in the required libraries are also restricted to the ASCII subset of Unicode and the character $\pi$.

## 3.1 Layout

Grace uses curly brackets for grouping, and semicolons as statement terminators, and infers semicolons at the end of lines. Code layout cannot be inconsistent with grouping.

**code with punctuation:**

```
while {stream.hasNext} do {
    print(stream.read);
};
```

**code without punctuation:**

```
while {stream.hasNext} do {
    print(stream.read)
}
```

A line break followed by an increase in the indent level implies a line continuation, whereas line break followed by the next line at the same or lesser indentation implies a semicolon if one is permitted syntactically.

## 3.2 Comments

Grace's comments start with a pair of slashes // and are terminated by the end of the line, as in C++ and Java. Comments are *not* treated as whitespace. Each comment is conceptually attached to the smallest immediately preceding syntactic unit, except that comments following a blank line are attached to the largest immediately following syntactic unit.

```
// comment, to end of line
```

### 3.3 Identifiers

Identifiers must begin with a letter, which is followed by a sequence of zero or more letters, digits and prime"'" or underscore "_" characters.

A single underscore "_" acts as a placeholder identifier: it is treated as a fresh identifier everywhere it is used.

### 3.4 Reserved Words and Reserved Operators

Grace has the following reserved words and reserved operators. The ? indicates words related to design options not yet chosen.

**class def inherits is method object**
**outer prefix return self Selftype**(?) **super type var where**

. := = ; { } [ ] " ( ) : −>

### 3.5 Tabs and Control Characters

Newline can be represented by the unicode newline character, by carriage return, or by line feed; a line feed that immediately follows a carriage return is ignored.

Tabs and all other non-printing control characters (except carriage return, line feed, and line separator) are syntax errors, even in a string literal. (Escape sequences include special characters in string literals.)

## 4 Built-in Objects

### 4.1 Done

The object done of type Done is Grace's unit. Methods with no explicit result return done. Done has no methods except for asString and asDebugString.

### 4.2 Numbers

Grace supports a single type Number. Number supports at least 64-bit precision floats. Implementations may support other numeric types: a full specification of numeric types is yet to be completed.

Grace has three syntactic forms for numerals (that is, literals that denote Numbers).

1. Decimal numerals, written as strings of digits, optionally preceded by a minus.

2. Explicit radix numerals, written as a (decimal) number between 2 and 35 representing the radix, a leading x, and a string of digits, where the digits from 10 to 35 are represented by the letters A to Z, in either upper or lower case. As a special case, a radix of 0 is taken to mean a radix of 16. Explicit radix numerals may optionally be preceded by a minus.

3. Base-exponent numerals, always in decimal, which use e as the exponent indicator. Base-exponent numerals may optionally be preceded by a minus.

All literals evaluate to exact rational numbers; explicit conversions (such as f64) must be used to convert rationals to other objects.

**Examples**

```
1
−1
42
3.14159265
13.343e−12
−414.45e3
16xF00F00
2x10110100
0xdeadbeef // Radix zero treated as 16
```

## 4.3 Booleans

The predefined constants true and false denote the only two values of Grace's Boolean type. Boolean operators are written using && for and, || for or, and prefix ! for not.

**Examples**

```
P && Q
toBe || toBe.not
```

"Short circuit" (a.k.a non-commutative) boolean operators take blocks as their second argument:

**Examples**

```
P.andAlso { Q }
toBe.orElse { ! toBe }
```

### 4.4 Strings and Characters

String literals in Grace are written between double quotes. Strings literals support a range of escape characters such as `"\t\b"`, and also escapes for Unicode; these are listed in Table 1 Individual characters are represented by Strings of length 1. Strings are immutable Grace values (see §10) and so may be interned. Strings conform to the protocol of an immutable IndexableCollection, and Grace's standard library includes mechanisms to support efficient incremental string construction.

| Escape | Meaning | Escape | Meaning |
|---|---|---|---|
| \\ | backslash | \' | single quote |
| \" | double quote | \b | backspace |
| \n | line-feed | \r | carriage-return |
| \t | tab | \l | unicode NEL |
| \f | page down | \e | escape |
| \{ | left brace | \} | right brace |
| \ (\ space) | non-breaking space | | |

Table 1: Grace string escapes.

**Examples**

```
"Hello World!"
"\t"
"The End of the Line\n"
"A"
```

#### 4.4.1 String interpolation

Within a string literal, expressions enclosed in braces are treated specially. The expression is evaluated, the asString method is requested on the resulting object, and the resulting string is inserted into the string literal in place of the brace expression.

### Examples

```
"Adding {a} to {b} gives {a+b}"
```

## 5 Blocks

Grace blocks are lambda expressions; they may or may not have parameters. If a parameter list is present, the parameters are separated by commas and the list is terminated by the −> symbol.

```
{do.something}
{ i −> i + 1}
{ sum, next −> sum + next }
```

Blocks construct objects containing a method named apply, apply(n), apply(n, m), ... depending on the number of parameters. Requesting the apply method evaluates the block; it is an error to provide the wrong number of arguments.

### Examples

The looping construct

```
for (1..10) do {
    i −> print i
}
```

might be implemented as as method with a block parameter

```
method for (collection) do (block) {
    def stream = collection.iterator
    while {stream.hasNext} do {
        block.apply(stream.next)
    }
}
```

Here is another example:

```
var sum := 0
def summingBlock : Block<Number,Number> =
    { i:Number −> sum := sum + i }
summingBlock.apply(4)  // sum now 4
summingBlock.apply(32)  // sum now 36
```

Blocks are lexically scoped inside their containing method or block. A "naked" block literal, that is, a block literal that is neither the target of a method request nor an argument, is a syntax error.

The body of a block consists of a sequence of declarations and expressions. An empty body is allowed, and is equivalent to done.

# 6 Declarations

Declarations of constants and variables may occur anywhere within an object, a method, or a block: their scope is the whole of their defining object, method, or block. Grace has a single namespace for all identifiers; this shared namespace is used for methods, constants, variables and types. It is an error to declare a constant or variable that shadows a lexically-enclosing constant or variable. It is an error to attempt to redeclare anything in the same lexical scope.

## 6.1 Constants

Constant definitions are introduced using the **def** keyword; they bind an identifier to the value of an initialising expression, optionally at a precise type. Constants cannot be re-bound.

### Examples

```
def x = 3 * 100 * 0.01
def x:Number = 3  // means the same as the above
def x:Number      // Syntax Error: x must be initialised
```

## 6.2 Variables

Variable definitions are introduced using the **var** keyword; they optionally bind an identifier to the value of an initialising expression, optionally at a precise type. Variables can be re-bound to new values as often as desired, using an assignment statement. If a variable is declared without an initializing expression, it is said to be *uninitialized*; any attempt to access the value of an initialized variable is an error. This error may be caught either at run time or at compile time, depending on the cleverness of your implementor.

### Examples

```
var x:Rational := 3 // explicit type
var x:Rational // ok; x must be initialized before being referenced
var x := 3 // x has type Unknown
var x // x has Unknown type and uninitialised value
```

## 6.3 Methods

Methods are declared with the **method** keyword, a name, which may contain zero or more parameter lists interspersed through it, an optional return type declaration, and a method body. Methods define the action to be taken when th object containing the method receives a request with the given name. For this reason, methods may not be declared directly inside other methods.

```
method pi {3.141592634}

method greet(user)from(sender) {
    print ''{sender} sends his greetings, {user}.''
}

method either (a : Block0<Done>) or (b : Block0<Done>) -> Done {
    if (random.nextBoolean)
        then {a.apply} else {b.apply}
}
```

### 6.3.1 Returning a Value from a Method

Methods may contain one or more **return** e statements. If a **return** statement is executed, the method terminates with the value of the expression e. If the method declares a return type of Done, then no expression may follow the **return**. If execution reaches the end of the method body without executing a **return**, the method terminates and returns the value of the last expression evaluated. Thus, an empty method body returns done.

### 6.3.2 Method Names

Methods can be named by an identifier, or by a sequence of operator symbols. Methods can also be named by an identifier suffixed with ":="; this form of name is conventionally used for writer methods, both user-written and automatically-generated, as exemplified by value:= below. Prefix operator methods are named "**prefix**" followed by the operator character(s).

**Examples**

```
method +(other : Point) -> Point {
    (x +other.x) @ (y +other.y)
}

method + (other) {
    (x +other.x) @ (y +other.y)
}

method + (other)
    { return (x +other.x) @ (y +other.y) }

method value:=(n : Number) -> Done {
    print "value currently {value}, now assigned {n}"
    super.value:= n
}
```

### 6.3.3 Variable Arity Methods

Methods may have "repeated parameters"; this provides a way of defining a method of "variable arity", i.e., one that can be requested with different numbers or arguments. A repeated parameter, if present, must be the final parameter in any of the method's parameter lists. Repeated parameters are designated by a star "∗" before the name of the parameter. Inside the method, a repeated parameter has the type of a Sequence of the declared type — for example, a parameter strs declared ∗strs:String has the type Sequence<String>.

Grace does not provide a way of requesting a variable-arity method using a collection of values (rather than an explicit argument list). Library designers are therefore encouraged to first define methods that take collation arguments, and then to define variable-arity methods in terms of the methods with collection arguments.

**Examples**

```
method addAll(elements: Collection<T>) {
    for (elements) do { x ->
        if (! contains(x)) then {
            def t = findPositionForAdd(x)
```

```
                inner.at(t)put(x)
                size := size + 1
                ...
                }
            }
        }
        self
    }

    method add(*elements:T) { addAll(elements) }
```

#### 6.3.4 Type Parameters

Methods may be declared with type parameters; these type parameters may be constrained with **where** clauses.

**Examples**

```
method show(*strs : String) -> Done {
    for (strs) do { each -> print each }
}

method sumSq<T>(a : T, b : T) -> T where T <: Numeric {
    (a * a) + (b * b)
}

method prefix- -> Number
     { 0 - self }
```

## 7 Objects and Classes

Grace **object** constructor expressions and declarations produce individual objects. Grace provides **class** declarations to create classes of objects all of which have the same structure.

Grace's class and inheritance design is complete but tentative. We need experience before confirming the design.

### 7.1 Objects

Objects are created by object literals. The body of an object literal consists of a sequence of declarations.

```
object {
    def colour:Colour = Colour.tabby
    def name:String = "Unnamed"
    var miceEaten := 0
    method eatMouse {miceEaten := miceEaten + 1}
}
```

Object literals are lexically scoped inside their containing method, or block. In particular, any object-level expressions, initializer expressions and method bodies are executed  Each time an object literal is executed, a new object is created.

Object literals are expressions that evaluate to an object with the given features. Object literals can also contain statements, which are executed as a side-effect of evaluating the object literal. Thus, constant can be defined by an object literal, like this:

```
def unnamedCat = object {
      def colour : Colour = Colour.tabby
      def name : String = "Unnamed"
      var miceEaten := 0
      method eatMouse {miceEaten := miceEaten + 1 }
      print "The Cat {name} has been created."
}
```

to bind a name to an object. Repeated invocations of the reader method `unnamedCat` return the same object.

## 7.2 Classes

Class declarations combine the definition of a factory object with the definition of a single object creation method on that object. This method creates "instances of the class". A class declaration is syntactically a combination of an object literal and a method declaration; everything in the body of the class declaration can be imagined as being part of an implicit object literal that is executed each time an "instance of the class" is created.

### Examples

```
class aCat.ofColour ( c:Colour ) named (n: String) {
    def colour:Colour = c
    def name:String = n
    var miceEaten := 0
```

```
    method eatMouse {miceEaten := miceEaten + 1}
    print "The Cat {name} has been created."
}
```

This declares a class, binds it to the name aCat, and declares an instance creation method on that class called ofColour()named(). This method takes two arguments, and returns a newly-created object with the fields and methods listed. Creating the object also has the side-effect of printing the given string, since executable code in the class declaration is also part of the implicit object literal.

This class might be used as follows:

```
def fergus = aCat.ofColour(Colour.Tortoiseshell)named "Fergus"
```

This creates an object with fields colour (set to Colour.Tortoiseshell), name (set to "Fergus"), and miceEaten (initialized to 0), prints "The Cat Fergus has been created", and binds fergus to this object.

The above declaration for aCat is equivalent to the following nested object declarations:

```
def aCat = object {  // a cat factory
    method ofColour (c: Colour) named (n: String)
        object {   // the cat herself
            def colour:Colour = c
            def name:String = n
            var miceEaten := 0
            method eatMouse {miceEaten := miceEaten + 1}
            print "The cat {name} has been created."
        }
    }
}
```

If the programmer wants a factory object with more than one method, she is free to build such an object using nested object constructors.

## 7.3 Inheritance

Grace supports inheritance with "single subclassing, multiple subtyping" (like Java), by way of an **inherits** C clause in a class declaration or object literal.

A new declaration of a method can override an existing declaration, but overriding declarations must be annotated with **is** override. Overridden methods can be accessed via **super** calls (see §8.6). It is a static error for a field to override another field or a method. This example shows how a

subclass can override accessor methods for a variable defined in a superclass (in this case, to always return 0 and to ignore assignments).

```
class aPedigreeCat.ofColour (aColour) named (aName) {
    inherits aCat.ofColour (aColour) named (aName)
    var prizes := 0
    method miceEaten is override {0}
    method miceEaten:= (n:Number)->Number is override {return}
                                        // ignore attempts to change it
}
```

The right hand side of an **inherits** clause is restricted to be an expression that creates a new object, such as the name of a class followed by a request on its constructor method, or a request to copy an exiting object.

When executing inherited code, self is first bound to the object under construction, self requests are resolved in the same way as the finally constructed object, def and var initialisers and inline code are run in order from the topmost superclass down to the bottom subclass. Accesses to unitialised vars and defs raise uninitialised exceptions (§6.2).

## 7.4 Understanding Inheritance (under discussion)

Grace's class declarations can be understood in terms of a flattening translation to object constructor expressions that build the factory object. Understanding this translation lets expert programmers build more flexible factories.

The above declaration for **class** aPedigreeCat is broadly equivalent to the following nested object declarations, not considering types, modules, and *renaming superclass methods so that an object's method names must actually be unique.*

```
def aPedigreeCat = object { // a cat factory
    method ofColour (c: Colour) named (n: String) -> PedigreeCat {
        object { // the cat herself
            def colour : Colour := c
            def name : String := n
            var Cat__miceEaten := 0 // ugly. very ugly
            var prizes = 0
            method miceEaten {0}
            method miceEaten:=(n:Number) {return} // ignore attempts to change it
        } // object
    } // method
```

```
} // object
```

### 7.5 Parameterized Classes

Classes may optionally be declared with type parameters. The corresponding requests on the factory methods may optionally be provided with type arguments. Type parameters may be constrained with **where** clauses.

**Examples**

```
class aVector.ofSize(size)<T> {
    var contents := Array.size(size)
    method at(index : Number) -> T {return contents.at() }
    method at(index : Number) put(elem : T) { }
}

class aSortedVector.ofSize<T>
    where T <: Comparable<T> {
        ...
}
```

## 8 Method Requests

Grace is a pure object-oriented language. Everything in the language is an object, and all computation proceeds by *requesting* an object to execute a method with a particular *name*. The response of the object is to execute the method. The value of a method request is the value returned by the execution of the method (see Section 6.3.1).

We distinguish the act of *requesting* a method (what Smalltalk calls "sending a message"), and *executing* that method. Requesting a method happens outside the object receiving the request, and involves only a reference to the receiver, the method name, and possibly some arguments. In contrast, executing the method involves the code of the method, which is local to the receiver.

### 8.1 Named Requests

A named method request is a receiver followed by a dot ".", then a method name (an identifier), then any arguments in parentheses. Parentheses are not used if there are no arguments. To improve readability, a the name of a

method that takes more than one parameter may comprise multiple parts or "words", with argument lists between the words, and following the last word. For example

```
method drawLineFrom(source)to(destination) { ... }
```

The name of a method and the position of its argument lists within that name is determined when the method is declared. When reading a request of a multi-part method name, you should continue accumulating words and argument lists as far to the right as possible.

If the receiver of a named method is **self** it may be left implicit, *i.e.*, the **self** and the dot may both be omitted.

Grace does not allow the "overloading" of method names: the type and number of arguments in a method request does not influence the name of the method that is being requested.

Parenthesis may be omitted where they would enclose a single argument that is a string or block literal.

**Examples**

```
canvas.drawLineFromPoint(p1)toPoint(p1)
canvas.drawLineFromPoint(origin)ofLenthXY(3,5)
canvas.movePenToXY(x,y)
canvas.movePenToPoint(p)
print "Hello world"
size
```

## 8.2 Assignment Requests

An assignment request is a variable followed by ":=", or it is a request of a method whose name ends with :=, in each case followed by a single argument. Spaces are optional before and after the ":=".

**Examples**

```
x := 3
y:=2
widget.active := true
```

Assignment methods return Done.

## 8.3 Binary Operator Requests

Binary methods have a receiver and one argument. Grace allows operator symbols to name binary methods, but the receiver of a binary method must always be explicit.

A binary operator method is named by one or more operator characters, provided that the operator is not a reserved symbol of the Grace language. So, for example, +, ++ and .. are valid operator symbols, but . is not, because it is reserved.

Most Grace operators have the same precedence: it is a syntax error for two different operator symbols to appear in an expression without parenthesis to indicate order of evaluation. The same operator symbol can be requested more than once without parenthesis; such expressions are evaluated left-to-right.

Four simple arithmetic operators do have precedence: / and ∗ over + and −.

**Examples**

```
1 + 2 + 3       // evaluates to 6
1 + (2 * 3)     // evaluates to 7
(1 + 2) * 3     // evaluates to 9
1 + 2 * 3       // evaluates to 7
1 +*+ 4 -*- 4   // syntax error
```

Named method requests without arguments bind more tightly than operator method requests. The following examples show the Grace expressions on the left, and the parse on the right.

**Examples**

| | |
|---|---|
| 1 + 2.i | 1 + (2.i) |
| (a ∗ a) + (b ∗ b).sqrt | (a ∗ a) + ((b ∗b).sqrt) |
| ((a ∗ a) + (b ∗ b)).sqrt | ((a ∗ a) + (b ∗b)).sqrt |
| a ∗ a + b ∗ b | (a ∗ a) + (b ∗b) |
| a + b + c | (a + b) + c |
| a − b − c | (a − b) − c |

## 8.4 Unary Prefix Operator Requests

Grace supports unary methods named by operator symbols that precede the explicit receiver. (Since binary operator methods must also have an explicit

receiver, there is no syntactic ambiguity.)

Prefix operators bind less tightly than other method requests, and therefore need parentheses to disambiguate.

**Examples**

```
−3 + 4
(−b).squared
−(b.squared)
− b.squared // parses as −(b.squared)

status.ok := !engine.isOnFire && wings.areAttached && isOnCourse
```

## 8.5 Accessing Operator Requests

Grace supports operators [...] and [...]:=, which can be defined in libraries, *e.g.*, for indexing and modifying collections.

**Examples**

```
print( a[3] ) // requests method [] on a with argument 3
a[3] := "Hello" // requests method []:= on a with arguments 3 and "Hello"
```

## 8.6 Super Requests

The reserved word **super** may be used only as an explicit receiver. In overriding methods, method requests with the pseudo-receiver **super** request the prior overridden method with the given name from **self**. Note that no "search" is involved; super-requests can be resolved statically, unlike other method requests.

**Examples**

```
super.value
super.bar(1,2,6)
super.doThis(3) timesTo("foo")
super + 1
!super

foo(super) // syntax error
1 + super // syntax error
```

## 8.7 Outer

The reserved word **outer** is used to refer to identifiers in lexically enclosing scopes. The expression **outer**.x refers to the innermost lexically enclosing identifier x; it is an error if there is no such x. If there are multiple enclosing declarations of x, then only the innermost is accessible; if a programmer finds it necessary to refer to one of the others, then the programmer should change the name to avoid this problem.

►NOTE: minigrace *currently recognizes* ***outer*** *as a* method *that can be requested of any object and that answers a reference to its enclosing object. This is a known limitation.*◄

### Examples

```
outer   // illegal
outer.value
outer.bar(1,2,6)
outer.outer.doThis(3) timesTo("foo") // illegal
outer + 1 // illegal (requests the binary + method, but on what receiver?)
!outer    // illegal (requests the prefix ! method, but on what receiver?)
```

## 8.8 Encapsulation

Grace has different default encapsulation rules for methods, types, and fields. The defaults can be changed by explicit annotations. The details are as follows.

### 8.8.1 Methods and Types

By default, methods and types are public, which means that they can be requested by any client that has access to the object. Thus, any expression can be the target of a request for a public method.

If a method or type is annotated **is** confidential, it can be requested only on the target **self** or **super**. This means that such a method or type is accessible to the object that contains it, and to inheriting objects, but not to client objects.

Methods and Types can be explicitly annotated as **is** public; this has no effect unless a dialect changes the default encapsulation.

Some other languages support "private methods", which are available only to an object itself, and not to clients or subobjects. Grace has neither private methods nor private types.

### 8.8.2 Fields

Variables and definitions (**var** and **def** declarations) immediately inside an object constructor create *fields* in that object.

A field declared as **var** x can be read using the request x and assigned to using the assignment request x:=() (see §8.2). A field declared as **def** y can be read using the request y, and cannot be assigned. By default, fields are *confidential*: they can be accessed and assigned from the object itself, and inheriting objects, and from lexically-enclosed objects, but not from clients. In other words, these requests can be made only on **self** and **super** and **outer**. However, this default visibility can be changed using annotations.

The annotation readable can be applied to a **def** or **var** declaration, and makes the accessor request available to any object. The annotation writable can be applied to a **var** declaration, and makes the assignment request available to any object. It is also possible to annotate a field declaration as public. In the case of a **def**, public is equivalent to (and preferred over) readable. In the case of a **var**, public is equivalent to readable, writable.

Fields and methods share the same namespace. The syntax for variable access is identical to that for requesting a reader method, while the syntax for variable assignment is identical to that for requesting an assignment method. This means that an object cannot have a field and a method with the same name, and cannot have an assignment method x:=() as well as a **var** field x.

#### Examples

```
object {
    def a = 1                    // Confidential access to a
    def b is public = 2          // Public access to b
    def c is readable = 2        // Public access to c
    var d := 3                   // Confidential access and assignment
    var e is readable            // Public access and confidential assignment
    var f is writable            // Confidential access, public assignment
    var g is public              // Public access and assignment
    var h is readable, writable  // Public access and assignment
}
```

### 8.8.3 No Private Fields

Some other languages support "private fields", which are available only to an object itself, and not to clients or inheritors. Grace does not have private fields; all fields can be accessed from subobjects. However, the parameters

and temporary variables of constructor methods can be used to obtain an effect similar to privacy.

**Examples**

```
method newShipStartingAt(s:Vector2D)endingAt(e:Vector2D) {
    // returns a battleship object extending from s to e. This object cannot
    // be asked its size, or its location, or how much floatation remains.
    assert ( (s.x == e.x) || (s.y == e.y) )
    def size = s.distanceTo(e)
    var floatation := size
    object {
        method isHitAt(shot:Vector2D) {
            if (shot.onLineFrom(s)to(e)) then {
                floatation := floatation −1
                if (floatation == 0) then { self.sink }
                true
            } else { false }
        }
        ...
    }
}
```

## 8.9 Requesting Methods with Type Parameters

Methods that have type parameters may be requested without explicit type arguments. When a method declared with type parameters is requested in a statically typed context without explicit type arguments, the type arguments are inferred.

**Examples**

```
sumSq<Integer64>(10.i64, 20.i64)
```

```
sumSq(10.i64, 20.i64)
```

## 8.10 Precedence of Method Requests

Grace programs are formally defined by the language's grammar (see appendix B). The grammar gives the following precedence levels; higher numbers bind more tightly.

1. Assignment operator := as a suffix to a named request or accessing operator.

2. "Other" operators; no priority for different operators; associate left to right.

3. "Additive" operators + and −; associate left to right.

4. "Multiplicative" operators ∗ and /; associate left to right.

5. Prefix operators; associate right to left.

6. Named requests (with or without arguments); the accessing operator. Multi-part named requests accumulate words and arguments as far to the right as possible.

7. Literals (numbers, strings, objects, types, ...); parenthesized expressions.

# 9 Control Flow

Control flow statements are requests to methods defined in the dialect. Grace uses what looks like conventional syntax with a leading keyword (if, while, for, etc.); these are actually method requests on the **outer** object defined in *standardGrace* or in some dialect.

## 9.1 Conditionals

```
if (test) then {block}

if (test) then {block} else {block}

if (test_1) then {block_1} elseif {test_2} then {block_2} ... else {block_n}
```

## 9.2 Looping statements

Grace has two bounded loops and an unbounded (while) loop.

**for statement:**

```
for (collection) do { each -> loop body }

for (course.students) do { s:Student -> print s }

for (0..n) do { i -> print i }
```

The first argument can be any object that answers an iterator when requested. Numeric ranges, collections and strings are typical examples. It is an error to modify the collection being iterated in the loop body. The block following do is executed repeatedly with the values yielded by the iterator as argument. Note that the block must have a single parameter; if the body of the block does not make use of the parameter, it may be named _.

**Examples**

```
for (1..4) do { _ -> turn 90; forward 10 }
```

For the common case where an action is repeated a fixed number of times, use repeat()times(), which takes a parameterless block:

```
repeat 4 times { turn 90; forward 10 }
```

**while statement:**

```
while {test} do {block}
```

Note that, since test can do a series of actions before returning a boolean, while()do() can be used to implement loops with exits in the middle or at the end, as well as loops with exits at the beginning.

## 9.3 Case

The match–case construct which attempts to match its first argument against a series of patterns introduced by case. Patterns support destructuring.

**Examples**

```
match (x)
    case { 0 -> "Zero" }
    // match against a literal constant
    case { s:String -> print(s) }
    // typematch, binding s - identical to block with typed parameter
```

```
case { (pi) -> print("Pi = " ++ pi) }
// match against the value of an expression - requires parenthesis
case { _ : Some(v) -> print(v) }
// typematch, binding a variable - looks like a block with parameter
case { _ -> print("did not match") }
// match against placeholder, matches anything
```

The case arguments are patterns: objects that understand the request match( ) and return a MatchResult, which is either a SuccessfulMatch object or a FailedMatch object. Each of the case patterns is requested to match(x) in turn, until one of them returns SuccessfulMatch(v); the result of the whole match–case construct is v.

### 9.3.1 Patterns

Pattern matching is based around the Pattern objects, which are objects that respond to a request match(anObject). The pattern tests whether or not the argument to match "matches" the pattern, and returns a MatchResult, which is either a SuccessfulMatch or a FailedMatch. An object that has type SuccessfulMatch behaves like the boolean true but also responds to the requests result and bindings. An object that has type FailedlMatch behaves like the boolean false but also responds to the requests result and bindings.

result is the return value, typically the object matched, and the bindings are a list of objects that may be bound to intermediate variables, generally used for destructuring objects.

For example, in the scope of this Point type:

```
type Point = {
  x -> Number
  y -> Number
  extract -> List<Number>
}
```

implemented by this class:

```
class aCartesianPoint.x(x':Number)y(y':Number) -> Point {
  method x { x' }
  method y { y' }
  method extract { aList.with(x, y) }
}
```

these hold:

```
def cp = aCartesianPoint.new(10,20)
```

```
Point.match(cp).result          // returns cp
Point.match(cp).bindings        // returns an empty list
Point.match(true)               // returns FailedMatch
```

### 9.3.2 Matching Blocks

Blocks with a single parameter are also patterns: they match any object that can validly be bound to that parameter. For example, if the parameter is annotated with a type, the block will successfully match an object that has that type, and will fail to match other objects.

Matching-blocks support an extended syntax for their parameters. In addition to being a fresh variable, as in a normal block, the parameter may also be a pattern. Matching-blocks are themselves patterns: one-argument (matching) block with parameter type A and return type R also implements Pattern<R,Done>.

A recursive, syntax-directed translation maps matching-blocks into blocks with separate explict patterns non-matching blocks that are called via apply only when their patterns match.

First, the matching block is flattened — translated into a straightforward non-matching block with one parameter for every bound name or placeholder. For example:

```
{ _ : Pair(a,Pair(b,c)) -> "{a} {b} {c}" }
```

is flattened into

```
{ _, a, b, c -> "{a} {b} {c}" }
```

then the pattern itself is translated into a composite object structure:

```
def mypat =
  MatchAndDestructuringPattern.new(Pair,
     VariablePattern.new("a"),
     MatchAndDestructuringPattern.new(Pair,
        VariablePattern.new("b"), VariablePattern.new("c")))
```

Finally, the translated pattern and block are glued together via a LambdaPattern:

```
LambdaPattern.new( mypat, { _, a, b, c -> "{a} {b} {c}" } )
```

The translation is as follows:

| e | [[ e ]] |
|---|---|
| _ : e | [[ e ]] |
| _ | WildcardPattern |
| v (fresh, unbound variable) | VariablePattern("v") |
| v (bound variable) | error |
| v : e | AndPattern.new(VariablePattern.new("v"),[[ e ]] ) |
| e(f,g) | MatchAndDestructuringPattern.new(e, [[f]], [[g]]) |
| literal | literal |
| e not otherwise translated | e |

### 9.3.3 Implementing Match-case

Finally the match(1)*case(N) methods can be implemented directly, e.g.:

```
method match(o : Any)
         case(b1 : Block<B1,R>)
         case(b2 : Block<B2,R>)
  {
    for [b1, b2] do { b ->
      def rv = b.match(o)
      if (rv.succeeded) then {return rv.result}
    }

    FailedMatchException.raise
  }
```

or (because matching-blocks are patterns) in terms of pattern combinators:

```
method match(o : Any)
         case(b1 : Block<B1,R>)
         case(b2 : Block<B2,R>)
  {
    def rv = (b1 || b2).match(o)
    if (rv.succeeded) then {return rv.result}

    FailedMatchException.raise
  }
```

**First Class Patterns** While all types are patterns, not all patterns are types. For example, it would seems sensible for regular expressions to be patterns, potentially created via one (or more) shorthand syntaxes (shorthands all defined in standard Grace)

```
match (myString)
  case { "" −> print "null string" }
  case { Regexp.new("[a−z]∗") −> print "lower case" }
  case { "[A−Z]∗".r −> print "UPPER CASE" }
  case { /"[0−9]∗" −> print "numeric" }
  case { ("Forename:([A−Za−z]∗)Surname:([A−Za−z]∗)".r2)(fn,sn) −>
              print "Passenger {fn.first} {sn}"}
```

With potentially justifiable special cases, more literals, e.g. things like tuples/lists could be descructured `[a,b,...] −>a ∗ b`. Although it would be very nice, it's hard to see how e.g. points created with `"3@4"` could be destructed like `a@b −>print "x: {a}, y: {b}"` without yet more bloated special-case syntax.

**Discussion** This rules try to avoid literal conversions and ambiguous syntax. The potential ambiguity is whether to treat something as a variable declaration, and when as a first-class pattern. These rules (should!) treat only fresh variables as intended binding instances, so a "pattern" that syntactically matches a simple variable declaration (as in this block `{ empty −>print "the singleton empty collection"}`) will raise an error — even though this is unambiguous given Grace's no shadowing rule.

Match statements that do nothing but match on types must distinguish themselves syntactically from a variable declaration, *e.g.*:

```
match (rv)
  case { (FailedMatch) −> print "failed" }
  case { _ : SuccessfulMatch −> print "succeeded" }
```

while writing just:

```
match (rv)
  case { FailedMatch −> print "failed" }
  case { SuccessfulMatch −> print "succeeded" }
```

although closer to the type declaration, less gratuitous, and perhaps less error-prone, would result in two errors about variable shadowing.

**Self-Matching** For this to work, the main value types in Grace, the main literals — Strings, Numbers — must be patterns that match themselves. That's what lets things like this work:

```
method fib(n : Number) {
  match (n)
    case { 0 -> 0 }
    case { 1 -> 1 }
    case { _ -> fib(n−1) + fib(n−2) }
}
```

With this design, there is a potential ambiguity regarding Booleans: "true || false" as an expression is very different from "true | false" as a composite pattern! Unfortunately, if Booleans are Patterns, then there's no way the type checker can distinguish these two cases.

If you want to match against objects that are not patterns, you can lift any object to a pattern that matches just that object by writing e.g. LiteralPattern.new(o) (option — or something shorter, like a prefix =~?).

## 9.4 Exceptions

Grace supports exceptions, which can be raised and caught. Exceptions are categorized into a hierarchy of ExceptionKindss.

At the site where an exceptional situation is detected, an exception is raised by requesting the raise method on an ExceptionKind object, with a string argument explaining the problem.

### Examples

```
BoundsError.raise "index {ix} not in range 1..{n}"
UserException.raise "Oops...!"
```

Exceptions are caught by a try(1)catch(1)...catch(1)finally() construct that syntactically parallels match(1)case(1)".

```
try {def f = file.open("data.store")}
    catch {e : NoSuchFile −> print("No Such File"); return}
    catch {e : PermissionError −> print("Permission denied"); return}
    catch {Exception −> print("Unidentified Error); system.exit}
    finally {f.close}
```

Raising an Exception terminates the execution of the expression containing the raise request; it is not possible to restart or resume that execution[1].

If an exception is raised in the try block, the catch blocks are attempted in order until one of them matches the exception. If none of them matches, then the process of matching the exception continues in the surrounding try(1)catch(1)...catch(1)finally(). The finally block is always executed before control leaves the try(1)catch(1)...catch(1)finally() construct, whether or not an exception is raised, or one of the catch blocks returns.

## 10 Equality and Value Objects

All objects automatically implement the following methods; programmers may override them.

1. == and != operators implemented as per Henry Baker's "egal" predicate [2]. That is, immutable objects are egal if they are of the same "shape", have the same methods declared in the same lexical environments, and if their fields' contents are egal, while mutable objects are only ever egal to themselves.

2. hashcode compatible with the egal.

As a consequence, immutable objects (objects with no **var** fields, which capture only other immutable objects) act as pure "value objects" without identity. This means that a Grace implementation can support value objects using whatever implementation is most efficient: either passing by reference always, by passing some times by value, or even by inlining fields into their containing objects, and updating the field if the containing object assigns a new value.

## 11 Types

Grace uses structural typing [11, 34, 17]. Types primarily describe the requests objects can answer. Fields do not directly influence types, except in so far as a field with publicly-visible accessor methods cause those methods to be part of the type (and in general to be visible to unconstrained clients).

[1] However, implementors should pickle the stack frames that are terminated when an exception is raised, so that they can be used in the error reporting machinery (debugger, stack trace)

Unlike in other parts of Grace, Type declarations are always statically typed, and their semantics may depend on the static types. The main case for this is determining between identifiers that refer to types, and those that refer to constant name definitions (introduced by **def**) which are interpreted as Singleton types.

## 11.1 Basic Types

Grace's standard prelude defines the following basic types:

- None — an uninhabited type. None conforms to all other types.
- Done — the type of the object returned by assignments and methods that have nothing interesting to return. All types conform to Done. The only methods on Done objects are asString and asDebugString
- Object — the common interface of most objects. It has methods ==, != (also written as or ≠), asString, asDebugString, and :: (binding construction). Objects that do not explicitly inherit from some other object implicitly inherit from a superobject with this type, and thus all objects (apart from done) have these methods, which will not be further mentioned.
- Boolean — the type of the objects true and false. Boolean has methods &&, ||, ==, ! =, ! (prefix-not), not, andAlso (short-circuit AND), orElse (short-circuit OR), and match.
- Number — the type of all numbers. Number has methods +, ∗, −, /, % (remainder), ^ (exponentiation), ++, <, <= (or ≤), >, >= (or ≥), .. (creating a range), − (prefix), inBase, truncate, and match.
- String — the type of character strings, and individual characters. String has methods ++, size, ord, at (also [ ]), iterator, substringFrom()to, replace()with, hash, indices, asNumber, indexOf, lastIndexOf, and match.
- Pattern — pattern used in match/case statements
- ExceptionKind — categorizing the various kinds of exceptional event. ExceptionKind has methods refine, raise, raise()with, match, |, & and parent.
- Exception — the type of a raised exception. Exception has methods message, lineNumber, moduleName, backtrace, printBackTrace, data and exception.

In addition, variables can be annotated as having type Unknown. Unknown is not a type, but a label that the type system uses when reasoning about the values of expressions. Parameters and variables that lack explicit type annotations are implicitly annotated with type Unknown.

## 11.2 Types

Types define the interface of objects by detailing their public methods, and the types of the arguments and results of those methods. Types can also contain definitions of other types.

The various Cat object and class descriptions (see §7) would produce objects that conform to an object type such as the following. Notice that the public methods implicitly inherited from Object are implicitly included in all types.

```
type {
    colour -> Colour
    name -> String
    miceEaten -> Number
    miceEaten:= (n : Number) -> Done
}
```

For commonality with method declarations, parameters are normally named in type declarations. These names are useful when writing specifications of the methods. If a parameter name is omitted, it must be replaced by an underscore. The type of a parameter may be omitted, in which case the type is Unknown.

## 11.3 Type Declarations

Types — and parameterized types — may be named in type declarations:

```
type MyCatType = { color -> Colour; name -> String }
   // I care only about names and colours

type MyParametricType<A,B> =
    where A <: Hashable, B <: DisposableReference
  type {
    at (_:A) put (_:B) -> Boolean
    cleanup(_:B)
  }
```

Notice that the **type** keyword may be omitted from the right-hand-side of a type declaration when it is a simple type literal.

Grace has a single namespace: types live in the same namespace as methods and variables.

```
type MyParametricType = <A,B>
{
    at (_:A) put (_:B) -> Boolean
    cleanup(_:B)
}
```

## 11.4 Relationships between Types — Conformance Rules

The key relation between types is **conformance**. We write B <: A to mean B conforms to A; that is, that B has all of the methods of A, and perhaps additional methods (and that the corresponding methods have conforming signatures). This can also be read as "B is a subtype of A", "A is a supertype of B".

We now define the conformance relation more rigorously. This section draws heavily on the wording of the Modula-3 report [11].

If B <: A, then every object of type B is also an object of type A. The converse does not apply.

If A and B are ground object types, then B <: A iff for every method m in A, there is a corresponding method $m$ (with the same name) in B such that

- The method $m$ in B must have the same number of arguments as $m$ in A, with the same distribution in multi-part method names.
- If the method $m$ in A has signature "$(P_1, ...P_n)$ -> $R$", and $m$ in B has signature "$(Q_1, ...Q_n)$ -> $S$", then
  - parameter types must be contravariant: $P_i <: Q_i$
  - results types must be covariant: S <: R

If a class or object B inherits from another class A, then B's type should conform to A's type. If A and B are parameterized classes, then similar instantiatons of their types should conform.

The conformance relationship is used in **where** clauses to constrain type parameters of classes and methods.

## 11.5 Variant Types

Variables with untagged, retained variant types, written T1 | T2 ... | Tn , may refer to an object of any one of their component types. No *objects* actually have variant types, only expressions. The actual type of an object referred to by a variant variable can be determined using that object's reified type information.

The only methods that may be requested via a variant type are methods with exactly the same declaration across all members of the variant. (Option) methods with different signatures may be requested at the most most specific argument types and least specific return type.

Variant types are retained as variants: they are *not* equivalent to the object type which describes all common methods. This is so that the exhaustiveness of match/case statements can be determined statically. Thus the rules for conformance are more restrictive:

```
S <: S | T; T <: S | T
(S' <: S) & (T' <: T) ==> (S' | T') <: (S | T)
```

The following example illustrates the limitations. Suppose

```
type S = {m: A -> B, n:C -> D}
type T = {m: A -> B, k: E -> F}
type U = {m: A -> B}
```

Then U fails to conform to S | T even though U contains all methods continued in both S and T.

## 11.6 Intersection Types

(option) An object conforms to an Intersection type, written T1 & T2 & ... & Tn, if and only if that object conforms to all of the component types. The main use of intersection types is as bounds on **where** clauses.

```
class Happy<T>
   where T <: (Comparable<T> & Printable & Happyable)
{  p ->
                  ...
}
```

## 11.7 Union Types

(option) Structural union types (sums), written T1 + T2 + ... + Tn, may refer to an object that conforms to any of the component types. Unions are

mostly included for completeness: variant types subsume most uses.

## 11.8 Type subtraction

(option) A type written T1 − T2 has the interface of T1 without any of the methods in T2.

## 11.9 Singleton Types

The names of singleton objects, typically declared in object declarations, may be used as types. Singleton types match only their singleton object. Singleton typs can be distinguised from regular types because Grace type declarations are statically typed.

```
def null = object {
    method isNull -> Boolean {return true}
}

type Some<T> {
    thing -> T
    isNull -> Boolean
}

type Option<T> = Some<T> | null
```

## 11.10 Nested Types

Type definitions may be nested inside other expressions, for example, they may be defined inside object, class, method, and other type definitions. Such types can be referred to using "dot" notation, written o.T. This allows a type to be used as a specification module, and for types to be imported from modules, since modules are objects.

## 11.11 Additional Types of Types

option: Grace may support exact types (written =Type)

option: Grace probably will probably not support Tuple types, probably written Tuple<T1, T2, ..., Tn>.

option: Grace may support selftypes, written **Selftype**.

## 11.12 Syntax for Types

This is very basic - but hopefully better than nothing!

```
Type := GroundType | (Type ("|" | "&" | "+") Type) | "(" Type ")"
GroundType ::= BasicType | BasicType "<" Type ","... ">" | "Selftype"
BasicType ::= TypeID | "=" TypeID
```

## 11.13 Reified Type Information Metaobjects and Type Literals

(option) Types are represented by objects of type Type (Hmm, should be Type<T>?). Since Grace has a single namespace, types can be accessed by requesting their names.

To support anonymous type literals, types may be written in expressions: **type** Type. This expression returns the type metaobject representing the literal type.

## 11.14 Type Assertions

(option) Type assertions can be used to check conformance and equality of types.

```
assert {B <: A}
    // B 'conforms to' A.
assert {B <: {foo(_:C) −> D} }
    // B had better have a foo method from C returning D
assert {B = A | C}
```

## 11.15 Notes

1. (**Sanity Check**) these rules

2. What's the relationship between "type members" across inheritance (and subtyping???). What are the rules on methods etc.

3. On matching, How does destructuring match works? What's the protocol? Who defines the extractor method? (not sure why this is here)

4. can a type extend another type?

5. Structural typing means we neither need nor want any variance annotations! Because Grace is structural, programmers can always write an (anonymous) structural type that gives just the interface they need — or such types could be stored in a library.

6. ObjectTypes require formal parameter names & need to fix examples. §11.2?

7. Tuples §11.11. Syntax as a type? Literal Tuple Syntax?

8. Nesting.

9. Serialization

10. Include dialect description.

# 12 Pragmatics

The distribution medium for Grace programs, objects, and libraries is Grace source code.

Grace source files should have the file extension .grace. If, for any bizzare reason a trigraph extension is required, it should be .grc

Grace files may start with one or more lines beginning with "#": these lines are ignored.

## 12.1 Garbage Collection

Grace implementations should be garbage collected. Safepoints where GC may occur are at any backwards branch and at any method request.

Grace will not support finalisation.

## 12.2 Concurrency and Memory Model

The core Grace specification does not describe a concurrent language. Different concurrency models may be provided as dialects.

Grace does not provide overall sequential consistency. Rather, Grace provides sequential consistency within a single thread. Across threads, any value that is read has been written by some thread sometime — but Grace does not provide any stronger guarantee for concurrent operations that interfere.

Grace's memory model should support efficient execution on architectures with Total Store Ordering (TSO).

# 13 Libraries

## 13.1 Collections

Grace will support some collection classes.

Collections will be indexed 1..size by default; bounds should be able to be chosen when explicitly instantiating collection classes.

# Acknowledgements

Thanks to Josh Bloch, Cay Horstmann, Micahel Kølling, Doug Lea, the participants at the Grace Design Workshops and the IFIP WG2.16 Programming Language Design for discussions about the language design.

Thanks to Michael Homer and Ewan Tempero for their comments on drafts.

The Scala language specification 2.8 [39] and the Newspeak language specification 0.05 [6] were used as references for early versions of this document. The design of Grace (so far!) has been influenced by Algol [41, 38], AmbientTalk [12], AspectJ [29], BCPL [42, **?**], Beta [33], Blue [30, 31, 32], C [28], C++ [44], C♯ [4, 3], Dylan [43], Eiffel [35, 36], Emerald [5], $F_1$ [10], $F\sharp$ [46], $FGJ$ [24], $FJ\vee$ [25], FORTRESS [1], gBeta [14], Haskell [22], Java [13, 18], Kevo [47], Lua [23], Lisp [16], ML [37], Modula-2 [51], Modula-3 [11], Modular Smalltalk [50], Newspeak [8, 6], Pascal [27], Perl [49], Racket [15], Scala [40, 39], Scheme [45], Self [48], Smalltalk [19, 26, 9, 7], Object-Oriented Turing [21], Noney [34], and Whiteoak [17] at least: we apologise if we've missed any languages out. All the good ideas come from these languages: the bad ideas are all our fault [20].

# A To Be Done

As well as the large list in Section 1 of features we haven't started to design, this section lists details of the language that remain to be done:

1. specify full numeric types
2. Block::apply §5 — How should we spell "apply"? "run"?
3. confirm method lookup algorithm, in particular relation between lexical scope and inheritance (see §8) ("Out then Up"). Is that enough? Does the no-shadowing rule work? If it does, is this a problem?
4. confirm "super" or other mechanism for requesting overridden methods §8.6.
5. confirm "outer" as meaning "the object enclosing self" §8.7
6. update grammar to incude "outer" §8.7.
7. confirm rules on named method argument parenthesization §8.1
8. how are (mutually) recursive names initialised?
9. make the **def** keyword optional, or remove it, or return to **const** §6.1
10. support multiple constructors for classes §7.2
11. where should we draw the lines between object constructor expressions/named object declarations, class declarations, and "hand-built" classes? §7.3
12. what's the difference between **class** FOO {} and **def** FOO = **class** {} (for various values of "class")
13. how do factories etc relate to "uninitialized" §6.2
14. decide what to do about equality operators §10
15. Support for enquiring about static type( decltype ?) and dynamic type (dyntype ?). Note that neither of these is a method request.
16. What is the type system?
17. Multiple Assignment §6.2

18. Type assertions — should they just be normal assertions between types? so *e.g.*, <: could be a normal operator between types.

19. Grace needs subclass compatibility rules

20. BRANDS. Brand Brand Brand.

21. weak references

22. virtualise literals — numbers, strings,

23. Do we want a built-in sequence constructor, or tuple constructor?

24. design option — generalised requests, that is, requests with 0 or more repeating parts like *elseif*

# B Grammar

```
// top level
def program = rule {codeSequence ~ rep(ws) ~ end}
def codeSequence = rule { repdel((declaration | statement), semicolon) }
def innerCodeSequence = rule { repdel((innerDeclaration | statement), semicolon) }


// declarations

def declaration = rule { varDeclaration | defDeclaration | classDeclaration |
                         typeDeclaration | methodDeclaration }
def innerDeclaration = rule { varDeclaration | defDeclaration | classDeclaration |
                         typeDeclaration }

def varDeclaration = rule { varId ~ identifier ~ opt(colon ~ typeExpression) ~
                        opt(assign ~ expression) }
def defDeclaration = rule { defId ~ identifier ~ opt(colon ~ typeExpression) ~
                       equals ~ expression }
def methodDeclaration = rule { methodId ~ methodHeader ~ methodReturnType ~ whereClause ~
                         lBrace ~ innerCodeSequence ~ rBrace }
def classDeclaration = rule { classId ~ identifier ~ dot ~ classHeader ~ methodReturnType ~ whereClause ~
                         lBrace ~ inheritsClause ~ codeSequence ~ rBrace }

//def oldClassDeclaration = rule { classId ~ identifier ~ lBrace ~
// opt(genericFormals ~ blockFormals ~ arrow) ~ codeSequence ~ rBrace }


//warning: order here is significant!
def methodHeader = rule { accessingAssignmentMethodHeader | accessingMethodHeader |
                         assignmentMethodHeader |
                         methodWithArgsHeader | unaryMethodHeader | operatorMethodHeader |
                         prefixMethodHeader }

def classHeader = rule { methodWithArgsHeader | unaryMethodHeader }
```

```
def inheritsClause = rule { opt( inheritsId ~ expression ~ semicolon ) }

def unaryMethodHeader = rule { identifier ~ genericFormals }
def methodWithArgsHeader = rule { firstArgumentHeader ~ repsep(argumentHeader,opt(ws)) }
def firstArgumentHeader = rule { identifier ~ genericFormals ~ methodFormals }
def argumentHeader = rule { identifier ~ methodFormals }
def operatorMethodHeader = rule { otherOp ~ oneMethodFormal }
def prefixMethodHeader = rule { opt(ws) ~ token("prefix") ~ otherOp } // forbid space after prefix?
def assignmentMethodHeader = rule { identifier ~ assign ~ oneMethodFormal }
def accessingMethodHeader = rule { lrBrack ~ genericFormals ~ methodFormals }
def accessingAssignmentMethodHeader = rule { lrBrack ~ assign ~ genericFormals ~ methodFormals }

def methodReturnType = rule { opt(arrow ~ nonEmptyTypeExpression ) }

def methodFormals = rule { lParen ~ rep1sep( identifier ~ opt(colon ~ typeExpression), comma) ~ rParen}
def oneMethodFormal = rule { lParen ~ identifier ~ opt(colon ~ typeExpression) ~ rParen}
def blockFormals = rule { repsep( identifier ~ opt(colon ~ typeExpression), comma) }

def matchBinding = rule{ (identifier | literal | parenExpression) ~
                              opt(colon ~ nonEmptyTypeExpression ~ opt(matchingBlockTail)) }
def matchingBlockTail = rule { lParen ~ rep1sep(matchBinding, comma) ~ rParen }

def typeDeclaration = rule { typeId ~ identifier ~ genericFormals ~
                                 equals ~ nonEmptyTypeExpression ~ semicolon ~ whereClause}

def typeExpression = rule { (opt(ws) ~ typeOpExpression ~ opt(ws)) | opt(ws) }
def nonEmptyTypeExpression = rule { opt(ws) ~ typeOpExpression ~ opt(ws) }

def typeOp = rule { opsymbol("|") | opsymbol("&") | opsymbol("+") }

// def typeOpExpression = rule { rep1sep(basicTypeExpression, typeOp) }

def typeOpExpression = rule { // this complex rule ensures two different typeOps have no precedence
  var otherOperator
  basicTypeExpression ~ opt(ws) ~
    opt( guard(typeOp, { s -> otherOperator:= s; true }) ~
         rep1sep(basicTypeExpression ~ opt(ws),
             guard(typeOp, { s -> s == otherOperator })
         )
    )
  }

def basicTypeExpression = rule { nakedTypeLiteral | literal | pathTypeExpression | parenTypeExpression }
   // if we keep this, note that in a typeExpression context { a; } is interpreted as type { a; }
   // otherwise as the block { a; }

def pathTypeExpression = rule { opt(superId ~ dot) ~ rep1sep((identifier ~ genericActuals),dot) }

def parenTypeExpression = rule { lParen ~ typeExpression ~ rParen }



// statements

def statement = rule { returnStatement | (expression ~ opt(assignmentTail)) }
    // do we need constraints here on which expressions can have an assignmentTail
```

```
    // could try to rewrite as options including (expression ~ arrayAccess ~ assignmentTail)
    // expression ~ dot ~ identifier ~ assignmentTail

def returnStatement = rule { symbol("return") ~ opt(ws) ~ opt(expression) } //doesn't need parens
def assignmentTail = rule { assign ~ expression }

// expressions

def expression = rule { opExpression }

//def opExpression = rule { rep1sep(addExpression, otherOp)}

def opExpression = rule { // this complex rule ensures two different otherOps have no precedence
  var otherOperator
  addExpression ~ opt(ws) ~
    opt( guard(otherOp, { s -> otherOperator:= s; true }) ~
        rep1sep(addExpression ~ opt(ws),
            guard(otherOp, { s -> s == otherOperator })
        )
    )
  }

def addExpression = rule { rep1sep(multExpression, addOp) }
def multExpression = rule { rep1sep(prefixExpression, multOp) }
def prefixExpression = rule { (rep(otherOp) ~ selectorExpression) | (rep1(otherOp) ~ superId) }
      // we can have !super

def selectorExpression = rule { primaryExpression ~ rep(selector) }

def selector = rule { (dot ~ unaryRequest) |
                      (dot ~ requestWithArgs) |
                      (lBrack ~ rep1sep(expression,comma) ~ rBrack)
                    }

def operatorChar = CharacterSetParser.new("!?@#$%^&|~=+-*/><:.") // had to be moved up

//special symbol for operators: cannot be followed by another operatorChar
method opsymbol(s : String) {trim(token(s) ~ not(operatorChar))}

def multOp = opsymbol "*" | opsymbol "/"
def addOp = opsymbol "+" | opsymbol "-"
def otherOp = rule { guard(trim(rep1(operatorChar)), { s -> ! parse(s) with( reservedOp ~ end ) })}
    // encompasses multOp and addOp
def operator = rule { otherOp | reservedOp }

def unaryRequest = rule { trim(identifier) ~ genericActuals ~ not(delimitedArgument) }
def requestWithArgs = rule { firstRequestArgumentClause ~ repsep(requestArgumentClause,opt(ws)) }
def firstRequestArgumentClause = rule { identifier ~ genericActuals ~ opt(ws) ~ delimitedArgument }
def requestArgumentClause = rule { identifier ~ opt(ws) ~ delimitedArgument }
def delimitedArgument = rule { argumentsInParens | blockLiteral | stringLiteral }
def argumentsInParens = rule { lParen ~ rep1sep(drop(opt(ws)) ~ expression, comma) ~ rParen
}

def implicitSelfRequest = rule { requestWithArgs | rep1sep(unaryRequest,dot) }

def primaryExpression = rule { literal | nonNakedSuper | implicitSelfRequest | parenExpression }
```

```
def parenExpression = rule { lParen ~ rep1sep(drop(opt(ws)) ~ expression, semicolon) ~ rParen }
                                        // TODO should parenExpression be around a codeSequence?

def nonNakedSuper = rule { superId ~ not(not( operator|lBrack )) }

// "generics"
def genericActuals = rule { opt(lGeneric ~ opt(ws) ~
                                rep1sep(opt(ws) ~ typeExpression ~ opt(ws),opt(ws) ~ comma ~ opt(ws)) ~
                                opt(ws) ~ rGeneric) }

def genericFormals = rule { opt(lGeneric ~ rep1sep(identifier, comma) ~ rGeneric) }

def whereClause = rule { repdel(whereId ~ typePredicate, semicolon) }
def typePredicate = rule { expression }

//wherever genericFormals appear, there should be a whereClause nearby.


// "literals"

def literal = rule { stringLiteral | selfLiteral | blockLiteral | numberLiteral |
                     objectLiteral | tupleLiteral | typeLiteral }

def stringLiteral = rule { opt(ws) ~ doubleQuote ~ rep( stringChar ) ~ doubleQuote ~ opt(ws) }
def stringChar = rule { (drop(backslash) ~ escapeChar) | anyChar | space}
def blockLiteral = rule { lBrace ~ opt( (matchBinding | blockFormals) ~ arrow)
                                  ~ innerCodeSequence ~ rBrace }
def selfLiteral = symbol "self"
def numberLiteral = trim(DigitStringParser.new)
def objectLiteral = rule { objectId ~ lBrace ~ inheritsClause ~ codeSequence ~ rBrace }
def tupleLiteral = rule { lBrack ~ repsep( expression, comma ) ~ rBrack }

def typeLiteral = rule { typeId ~ opt(ws) ~ nakedTypeLiteral }
def nakedTypeLiteral = rule { lBrace ~ opt(ws) ~
                                repdel(methodHeader ~ methodReturnType, (semicolon | whereClause)) ~
                                opt(ws) ~ rBrace }

// terminals
def backslash = token "\\" // doesn't belong here, doesn't work if left below!
def doubleQuote = token "\""
def space = token " "
def semicolon = rule { (symbol(";") ~ opt(trim(newLine))) }
def colon = rule {both(symbol ":",not(assign))}
def newLine = symbol "\n"
def lParen = symbol "("
def rParen = symbol ")"
def lBrace = symbol "\{"
def rBrace = symbol "\}"
def lBrack = symbol "["
def rBrack = symbol "]"
def lrBrack = symbol "[]"
def arrow = symbol "->"
def dot = symbol "."
def assign = symbol ":="
def equals = symbol "="
```

```
def lGeneric = token "<"
def rGeneric = token ">"

def comma = rule { symbol(",") }
def escapeChar = CharacterSetParser.new("\\\"'\{\}bnrtlfe ")

def azChars = "abcdefghijklmnopqrstuvwxyz"
def AZChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
def otherChars = "1234567890~!@#$%^&*()_-+=[]|\:;<,>.?/"

def anyChar = CharacterSetParser.new(azChars ++ AZChars ++ otherChars)

def identifierString = trim(GraceIdentifierParser.new)

// def identifier = rule { bothAll(trim(identifierString),not(reservedIdentifier)) }
                              // bothAll ensures parses take the same length
// def identifier = rule{ both(identifierString,not(reservedIdentifier)) }
                             // both doesn't ensure parses take the same length
def identifier = rule { guard(identifierString, { s -> ! parse(s) with( reservedIdentifier ~ end ) })}
                            // probably works but runs out of stack

// anything in this list needs to be in reservedIdentifier below (or it won't do what you want)
def superId = symbol "super"
def extendsId = symbol "extends"
def inheritsId = symbol "inherits"
def classId = symbol "class"
def objectId = symbol "object"
def typeId = symbol "type"
def whereId = symbol "where"
def defId = symbol "def"
def varId = symbol "var"
def methodId = symbol "method"
def prefixId = symbol "prefix"
def interfaceId = symbol "interface"

def reservedIdentifier = rule {selfLiteral | superId | extendsId | inheritsId |
            classId | objectId | typeId | whereId |
            defId | varId | methodId | prefixId | interfaceId } // more to come

def reservedOp = rule {assign | equals | dot | arrow | colon | semicolon} // this is not quite right
```

## References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0$\beta$. Technical report, Sun Microsystems, Inc., March 2007.

[2] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), October 1993.

[3] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C♯. In *OOPSLA*, 2007.

[4] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C♯. In *ECOOP*, 2010.

[5] Andrew P. Black, Eric Jul, Norman Hutchinson, and Henry M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*. ACM Press, 2007.

[6] Gilad Bracha. Newspeak programming language draft specification version 0.0. Technical report, Ministry of Truth, 2009.

[7] Gilad Bracha and David Griswold. Stongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*. ACM Press, 1993.

[8] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda6. Modules as objects in Newspeak. In *ECOOP*, 2010.

[9] Tim Budd. *A Little Smalltalk*. Addison-Wesley, 1987.

[10] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *Computer Science Handbook*, chapter 97. CRC Press, 2nd edition, 2004.

[11] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 reference manual. Technical Report Research Report 53, DEC Systems Research Center (SRC), 1995.

[12] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In *ECOOP*, pages 230–254, 2006.

[13] Carlton Egremont III. *Mr. Bunny's Big Cup o' Java*. Addison-Wesley, 1999.

[14] Erik Ernst. Family polymorphism. In *ECOOP*, 2001.

[15] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How To Design Programs*. MIT Press, 2001.

[16] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, 1991.

[17] Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA*, 2008.

[18] Brian Goetz, Time Peierls, Joshua Block, Joesph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[19] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[20] C.A.R. Hoare. Hints on programming language design. Technical Report AIM-224, Stanford Artificial Intelligence Laboratory, 1973.

[21] Ric Holt and Tom West. OBJECT ORIENTED TURING REFERENCE MANUAL seventh edition version 1.0. Technical report, Holt Software Associates Inc., 1999.

[22] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *History of Programming Languages III*, pages 12–1–12–55. ACM Press, 2007.

[23] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL-III*, 2007.

[24] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[25] Atsushi Igarashi and Hideshi Nagira. Union types for object- oriented programming. *Journal of Object Technology*, 6(2):31–45, February 2007. http://www.jot.fm/issues/issues 2007 02/article3.

[26] Daniel H.H. Ingalls. Design principles behind Smalltalk. *BYTE Magazine*, August 1981.

[27] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer, 1975.

[28] Brian W. Kernighan and Dennis M. Ritchie. *The "C" Programming Language*. Addison-Wesley, 2nd edition, 1993.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[30] Michael Kölling, Bett Koch, and John Rosenberg. Requirements for a first year object-oriented teaching language. In *ACM Conference on Computer Science Education (SIGCSE)*, 1995.

[31] Michael Kölling and John Rosenberg. Blue — a language for teaching object-oriented programming. In *ACM Conference on Computer Science Education (SIGCSE)*, 1996.

[32] Michael Kölling and John Rosenberg. Blue — a language for teaching object-oriented programming language specification. Technical Report TR97-13, Monash University Department of Computer Science and Software Engineering, 1997.

[33] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[34] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP*, 2008.

[35] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[36] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[37] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[38] Peter Naur. The European side of the development of ALGOL. In *History of Programming Languages I*, pages 92–139. ACM Press, 1981.

[39] Martin Odersky. The Scala language specification version 2.8. Technical report, Programming Methods Laboratory, EFPL, July 2010.

[40] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[41] Alan J. Perlis. The American side of the development of ALGOL. In *History of Programming Languages I*, pages 75–91. ACM Press, 1981.

[42] Martin Richards and Colin Whitby-Stevens. *BCPL: the language and its compiler*. Cambridge University Press, 1980.

[43] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Apple Computer, Inc., 1996.

[44] Bjarne Stroustrup. Why C++ is not just an object-oriented programming language. In *OOPSLA Companion*. ACM Press, 1995.

[45] Gerald Sussman and Guy Steele. SCHEME: An interpreter for extended lambda calculus. Technical Report AI Memo 349, MIT Artificial Intelligence Laboratory, December 1975.

[46] Don Syme. The F♯ draft language specification. Technical report, Microsoft, 2009.

[47] Antero Taivalsaari. Delegation versus concatenation or cloning is inheritance too. *OOPS Messenger*, 6(3), 1995.

[48] David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

[49] Larry Wall. Perl, the first postmodern computer language. `http://www.wall.org/ larry/pm.html`, Spring 1999.

[50] Allen Wirfs-Brock and Brian Wilkerson. Modular Smalltalk. In *OOPSLA*, 1998.

[51] Niklaus Wirth. Modula-2 and Oberon. In *HOPL*, 2007.